Deductive Program Verification with WHY3

Andrei Paskevich

LRI, Université Paris-Sud — Toccata, Inria Saclay

ÉJCP 2016

1. A short look back

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explosion — an erroneous *float-to-int* conversion
- 1997: *Pathfinder* reset loop — priority inversion
- 1999: *Mars Climate Orbiter* explosion — a unit error

...

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explosion — an erroneous *float-to-int* conversion
- 1997: *Pathfinder* reset loop — priority inversion
- 1999: *Mars Climate Orbiter* explosion — a unit error

...

- 2006: *Debian SSH bug* — predictable RNG (fixed in 2008)
- 2012: *Heartbleed* — buffer over-read (fixed in 2014)
- 1989: *Shellshock* — insufficient input control (fixed in 2014)

...

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1962.

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1962.

2006: *Nearly All Binary Searches and Mergesorts are Broken*

(Joshua Bloch, Google, a blog post)

The code in JDK:

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1962.

2006: *Nearly All Binary Searches and Mergesorts are Broken*
(Joshua Bloch, Google, a blog post)

The code in JDK:

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

Bug: addition may exceed $2^{31} - 1$, the maximum `int` in Java.

One possible solution:

```
int mid = low + (high - low) / 2;
```

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

```
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
```

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

First theoretical foundation: Floyd-Hoare logic, 1969

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

First theoretical foundation: Floyd-Hoare logic, 1969

First grand success in practice: metro line 14, 1998

tool: Atelier B, proof by refinement

Other major success stories

- Flight control software in A380, 2005

  safety proof: the absence of execution errors

  tool: Astrée, abstract interpretation

- Hyper-V — a native hypervisor, 2008

  tools: VCC + automated prover Z3, deductive verification

- CompCert — certified C compiler, 2009

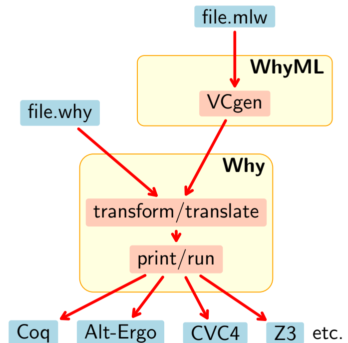
  tool: Coq, generation of the correct-by-construction code

- seL4 — an OS micro-kernel, 2009

  tool: Isabelle/HOL, deductive verification

2. Tool of the day

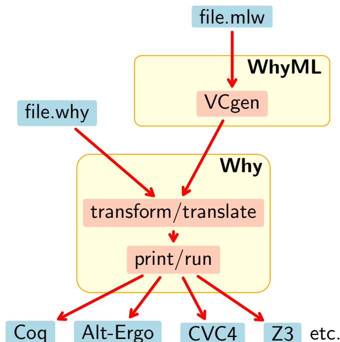
WHY3 in a nutshell



WHY3 in a nutshell

WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants



WHY3 in a nutshell

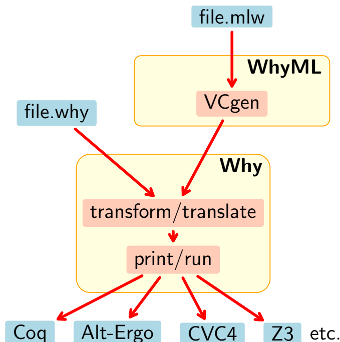
WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WHYML, a specification language

- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)



WHY3 in a nutshell

WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WHYML, a specification language

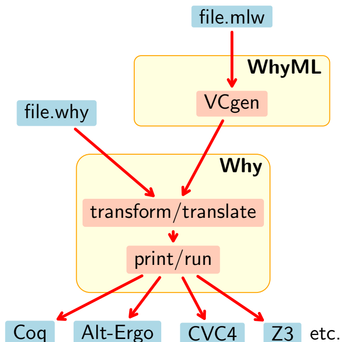
- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)

WHY3, a program verification tool

- VC generation using WP or fast WP
- 70+ VC *transformations* (≈ tactics)
- support for 25+ ATP and ITP systems

(Boogie 2011) (ESOP 2013) (VSTTE 2013)



WHY3 out of a nutshell

three different ways of using WHY3

- as a logical language
  - a convenient front-end to many theorem provers
- as a programming language to prove algorithms
  - see examples in our gallery
    http://toccata.lri.fr/gallery/why3.en.html
- as an intermediate verification language
  - Java programs: Krakatoa (Marché Paulin Urbain)
  - C programs: Frama-C (Marché Moy)
  - Ada programs: SPARK 2014 (Adacore)
  - probabilistic programs: EasyCrypt (Barthe et al.)

Example: maximum subarray problem

```
let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
```

Kadane's algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                      *)
(*  ..............................|### cur ####                      *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in




  for i = 0 to length a - 1 do





    cur += a[i];
    if !cur < 0 then       cur := 0;
    if !cur > !max then       max := !cur
  done;
  !max
```

Kadane's algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                     *)
(*  ..............................|### cur ####                     *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in


  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }


    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then       max := !cur
  done;
  !max
```

Kadane's algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                      *)
(*  ..............................|### cur ####                      *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Kadane's algorithm

```
use import ref.Refint
use import array.Array
use import array.ArraySum

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Why3 proof session

File View Tools Help

Context
- Unproved goals
- All goals

Strategies: Auto level 1, Auto level 2, Compute, Inline, Split

Provers: Alt-Ergo (1.01), CVC3 (2.2), CVC3 (2.4.1), CVC4 (1.0), Coq (8.5), Eprover (1.6), Spass (3.7), Z3 (2.19), Z3 (3.2), Z3 (4.0), Z3 (4.2)

Tools: Edit, Replay, Remove, Clean

Proof monitoring
Waiting: 0
Scheduled: 0
Running: 0
Interrupt

Theories/Goals	Status	Time
kadane.mlw	✓	0.20
Kadane	✓	0.20
VC for maximum_subarray	✓	0.20
split_goal_wp	✓	0.20
1. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps
2. postcondition	✓	0.00
Alt-Ergo (1.01)	?	0.00
Spass (3.7)	✓	0.00
3. loop invariant init	✓	0.00
4. loop invariant init	✓	0.00
5. loop invariant init	✓	0.00
6. loop invariant init	✓	0.00
7. index in array bounds	✓	0.00
8. loop invariant preservation	✓	0.02
9. loop invariant preservation	✓	0.01
10. loop invariant preservation	✓	0.02
11. loop invariant preservation	✓	0.00
12. loop invariant preservation	✓	0.02
13. loop invariant preservation	✓	0.01
14. loop invariant preservation	✓	0.02
15. loop invariant preservation	✓	0.00
16. loop invariant preservation	✓	0.02
17. loop invariant preservation	✓	0.01
18. loop invariant preservation	✓	0.02
19. loop invariant preservation	✓	0.01
20. loop invariant preservation	✓	0.02
21. loop invariant preservation	✓	0.00
22. loop invariant preservation	✓	0.02
23. loop invariant preservation	✓	0.00
24. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps
25. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps

Source code | Task | Edited proof | Prover Output | Counter-example

file: kadane/../kadane.mlw

```
1
2  (* Maximum subarray problem
3
4     Given an array of integers, find the continuous subarray with the
5     greatest sum. Subarrays of length 0 are allowed (which means that
6     an array with negative values only has a maximal sum of 0).
7
8     Authors: Jean-Christophe Filliâtre (CNRS)
9              Guillaume Melquiond       (Inria)
10             Andrei Paskevich          (UPSUD)
11  *)
12
13 module Kadane
14
15   use import int.Int
16   use import ref.Refint
17   use import array.Array
18   use import array.ArraySum
19
20 (*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |   *)
21 (*  ......|######## max ########|..............                        *)
22 (*  ..............................|### cur ####                        *)
23
24   let maximum_subarray (a: array int): int
25     ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
26     ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
27   =
28     let max = ref 0 in
29     let cur = ref 0 in
30     let ghost cl = ref 0 in
31     let ghost lo = ref 0 in
32     let ghost hi = ref 0 in
33     for i = 0 to a.length - 1 do
34       invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
35       invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
36       invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
37       invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
38       cur += a[i];
39       if !cur < 0 then begin cur := 0; cl := i+1 end;
40       if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
41     done;
42     !max
43
44 end
```

3. Program correctness • Weakest Precondition calculus

A simple language: pure terms

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} & \text{data types} \\
t & ::= & \ldots, -1, 0, 1, \ldots, 42, \ldots & \text{integer constants} \\
 & \mid & \texttt{true} \mid \texttt{false} & \text{Boolean constants} \\
 & \mid & \texttt{()} & \text{unit type constant} \\
 & \mid & x & \text{immutable variables} \\
 & \mid & \texttt{!}r & \text{pointer dereference} \\
 & \mid & t \; op \; t & \text{binary operations} \\
op & ::= & + \mid - \mid * & \text{arithmetic operations} \\
 & \mid & = \mid \neq \mid < \mid > \mid \leq \mid \geq & \text{arithmetic comparisons} \\
 & \mid & \wedge \mid \vee & \text{conjunction and disjunction}
\end{array}
$$

- mutable “references” (pointers) hold immutable data, are not terms
- well-typed terms evaluate without errors (no null pointers, no division)
- evaluation of a term does not modify the program memory

A simple language: expressions

$$
\begin{array}{lcll}
e & ::= & t & \text{pure term} \\
 & \mid & r := t & \text{assignment} \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e & \text{binding} \\
 & \mid & \texttt{let}\ r = \texttt{ref}\ t\ \texttt{in}\ e & \text{allocation} \\
 & \mid & \texttt{if}\ t\ \texttt{then}\ e\ \texttt{else}\ e & \text{conditional} \\
 & \mid & \texttt{while}\ t\ \texttt{do}\ e\ \texttt{done} & \text{loop}
\end{array}
$$

- expressions can modify memory (assignment)
- well-typed expressions evaluate without errors
- expressions may diverge: `while true do () done`
- no pointer aliasing: $\texttt{let}\ r' = r\ \texttt{in}\ \dots$ is not allowed

A simple language: syntactic sugar

$$\begin{array}{rcl}
e\ \texttt{;}\ e_1 & \equiv & \texttt{let}\ _ = e\ \texttt{in}\ e_1 \\
r := e & \equiv & \texttt{let}\ x = e\ \texttt{in}\ r := x \\
\texttt{let}\ r = \texttt{ref}\ e\ \texttt{in}\ e_1 & \equiv & \texttt{let}\ x = e\ \texttt{in}\ \texttt{let}\ r = \texttt{ref}\ x\ \texttt{in}\ e_1 \\
\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \equiv & \texttt{let}\ x = e\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
e_1\ \&\&\ e_2 & \equiv & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{false} \\
e_1\ \texttt{||}\ e_2 & \equiv & \texttt{if}\ e_1\ \texttt{then}\ \texttt{true}\ \texttt{else}\ e_2
\end{array}$$

Example

```
let sum = ref 1 in
let count = ref 0 in
while !sum <= n do
  count := !count + 1;
  sum := !sum + 2 * !count + 1
done;
!count
```

What is the result of this expression for a given n?

Example

```
let sum = ref 1 in
let count = ref 0 in
while !sum <= n do
  count := !count + 1;
  sum := !sum + 2 * !count + 1
done;
!count
```

What is the result of this expression for a given `n`?

Informal specification:

- at the end, `count` contains the truncated square root of `n`
- for instance, given `n = 42`, the returned value is 6

Hoare triples

A statement about program correctness:

$$\{P\}\ e\ \{Q\}$$

- P precondition property
- e expression
- Q postcondition property

Hoare triples

A statement about program correctness:

$$\{P\}\, e\, \{Q\}$$

- P precondition property
- e expression
- Q postcondition property

What is the meaning of a Hoare triple?

$\{P\}\, e\, \{Q\}$ is valid if and only if
when we start computing e in a state that satisfies P,
then the computation either does not terminate
or it terminates in a state that satisfies Q.

This is partial correctness: we do not prove termination.

Examples of valid Hoare triples for partial correctness:

- $\{!r = 1\}\ r := !r + 2\ \{!r = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do () done` $\{\texttt{false}\}$

In our square root example:

$$\{?\}\ \mathit{ISQRT}\ \{?\}$$

Examples of valid Hoare triples for partial correctness:

- $\{!r = 1\}$ r := $!r + 2$ $\{!r = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v : x = 4 * v\}$ $x + 42$ $\{\exists w : \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do () done` $\{\texttt{false}\}$

In our square root example:

$$\{n \geq 0\}\ \mathit{ISQRT}\ \{?\}$$

Examples

Examples of valid Hoare triples for partial correctness:

- $\{!r = 1\}\ r := \ !r + 2\ \{!r = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do () done` $\{\texttt{false}\}$

In our square root example:

$$\{n \geq 0\}\ \mathit{ISQRT}\ \{\texttt{result} * \texttt{result} \leq n < (\texttt{result} + 1) * (\texttt{result} + 1)\}$$

How can we establish correctness of a program?

One solution: EDSGER DIJKSTRA, 1975

Predicate transformer $\mathrm{WP}(e; Q)$

- e expression
- Q postcondition

computes the weakest precondition P such that $\{P\}\, e\, \{Q\}$

Definition of $\mathrm{WP}(e, Q)$

Recursive definition over the program structure:

$$
\begin{aligned}
\mathrm{WP}(t, Q) &= Q[\texttt{result} \leftarrow t] \\
\mathrm{WP}(r := t, Q) &= Q[!r \leftarrow t, \texttt{result} \leftarrow ()] \\
\mathrm{WP}(\texttt{let } x = e_0 \texttt{ in } e, Q) &= \mathrm{WP}(e_0, \mathrm{WP}(e, Q)[x \leftarrow \texttt{result}]) \\
\mathrm{WP}(\texttt{let } r = \texttt{ref } t \texttt{ in } e, Q) &= \mathrm{WP}(e, Q)[!r \leftarrow t] \\
\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) &= (t \rightarrow \mathrm{WP}(e_1, Q)) \wedge (\neg t \rightarrow \mathrm{WP}(e_2, Q))
\end{aligned}
$$

Below, we omit the $[\texttt{result} \leftarrow ()]$ substitution for unit-typed expressions.

Definition of $\mathrm{WP}(e, Q)$: loops

$$
\begin{aligned}
&\mathrm{WP}(\texttt{while}\ t\ \texttt{do}\ e\ \texttt{done}, Q) = \\
&\quad \exists J : \mathrm{Prop}. \\
&\qquad J \wedge \\
&\qquad \forall w_1, \ldots, w_k. \\
&\qquad\quad (J \wedge t \rightarrow \mathrm{WP}(e, J))[!\vec{r} \leftarrow \vec{w}] \wedge \\
&\qquad\quad (J \wedge \neg t \rightarrow Q)[!\vec{r} \leftarrow \vec{w}]
\end{aligned}
$$

some invariant property J
that holds at the loop entry
and is preserved
after a single iteration,
is strong enough to prove Q

$r_1, \ldots, r_k$ references modified in e

$w_1, \ldots, w_k$ fresh variables

We cannot know the values of modified references after n iterations

- therefore, we prove preservation and the post for arbitrary values
- the invariant must provide all the needed information about the state

Definition of $\mathrm{WP}(e, Q)$: annotated loops

Finding an appropriate invariant is difficult in the general case

- this is equivalent to constructing a proof of Q by induction

We can ease the task of automated tools by providing annotations:

$$\begin{aligned}
&\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{do}\ e\ \texttt{done}, Q) = \\
&\quad J \wedge \\
&\quad \forall w_1, \ldots, w_k. \\
&\quad\quad (J \wedge t \rightarrow \mathrm{WP}(e, J))[!\vec{r} \leftarrow \vec{w}] \wedge \\
&\quad\quad (J \wedge \neg t \rightarrow Q)[!\vec{r} \leftarrow \vec{w}]
\end{aligned}$$

the given invariant J holds at the loop entry, is preserved after a single iteration, and suffices to prove Q

$r_1, \ldots, r_k$ references modified in e

$w_1, \ldots, w_k$ fresh variables

Examples

$$\mathrm{WP}\big(x := \,!x + y;\, !x = 2y\big) \;\equiv\; !x + y = 2y$$

$$\mathrm{WP}(x := \texttt{!}x + y;\ \texttt{!}x = 2y) \equiv \texttt{!}x + y = 2y$$

$$\mathrm{WP}(\texttt{while } \texttt{!}y > 0 \texttt{ invariant } \mathit{pair}(\texttt{!}y) \texttt{ do } y := \texttt{!}y - 2 \texttt{ done};\ \mathit{pair}(\texttt{!}y)) \equiv$$

$$\mathrm{WP}(x := !x + y; !x = 2y) \equiv !x + y = 2y$$

$$\mathrm{WP}(\texttt{while } !y > 0 \texttt{ invariant } pair(!y) \texttt{ do } y := !y - 2 \texttt{ done}; \; pair(!y)) \equiv$$

$$pair(!y) \land$$
$$\forall v.(pair(v) \land v > 0 \to pair(v-2)) \land$$
$$\forall v.(pair(v) \land v \leq 0 \to pair(v))$$

Theorem

For any e and Q, the triple $\{\mathrm{WP}(e, Q)\}\ e\ \{Q\}$ is valid.

Can be proved by induction on the structure of program e
w.r.t. some reasonable semantics (axiomatic, operational, etc.)

Corollary

To show that $\{P\}\ e\ \{Q\}$ is valid, it suffices to prove $P \rightarrow \mathrm{WP}(e, Q)$.

This is what WHY3 does!

4. Safety properties: assertions and function calls

Certain operations can produce run-time errors
if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

Execution safety

Certain operations can produce run-time errors
if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

A correct program must not fail:

$\{P\}\,e\,\{Q\}$ is valid if and only if
when we start computing e in a state that satisfies P,
then the computation either does not terminate
or it terminates normally in a state that satisfies Q.

Assertions

A new expression:

$$e \quad ::= \quad \ldots \qquad | \quad \texttt{assert}\ R \qquad \text{fails if } R \text{ does not hold}$$

The corresponding weakest precondition rule:

$$\mathrm{WP}(\texttt{assert}\ R;\ Q) \quad = \quad R \wedge Q \quad = \quad R \wedge (R \rightarrow Q)$$

The second version is useful in practical deductive verification.

Calling subprograms

We may want to delegate some functionality to a function:

$$\texttt{let}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \mathscr{C} = e_f$$ defined function

$$\texttt{val}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \mathscr{C}$$ abstract function

The function behaviour is specified with a contract:

$\mathscr{C} ::=$	$\texttt{requires}\ P_f$	precondition
	$\texttt{writes}\ r_1, \ldots, r_k$	modified global references
	$\texttt{ensures}\ Q_f$	postcondition

Q_f may refer to the initial values of modified references: $r_1^{\circ}, \ldots, r_k^{\circ}$

e	$::=$	$\ldots$	
	$\mid$	$f\ t \ldots t$	function call

Verification condition for a function definition:

$$\mathrm{VC}(\texttt{let}\ f\ \ldots) \;=\; \forall w_1,\ldots,w_k,x_1,\ldots,x_n.\ (P_f \rightarrow \mathrm{WP}(e_f,Q_f)[\vec{r}^{\circ} \leftarrow !\vec{r}])[!\vec{r} \leftarrow \vec{w}]$$

- $x_1,\ldots,x_n$ formal parameters of f
- $r_1,\ldots,r_k$ references modified in e_f
- $r_1^{\circ},\ldots,r_k^{\circ}$ initial values of $r_1,\ldots,r_k$
- $w_1,\ldots,w_k$ fresh variables

Verification

Verification condition for a function definition:

$$\mathrm{VC}(\texttt{let}\ f \ldots) = \forall w_1, \ldots, w_k, x_1, \ldots, x_n.\ (P_f \rightarrow \mathrm{WP}(e_f, Q_f)[\vec{r}^{\circ} \leftarrow !\vec{r}])[!\vec{r} \leftarrow \vec{w}]$$

$x_1, \ldots, x_n$	formal parameters of f
$r_1, \ldots, r_k$	references modified in e_f
$r_1^{\circ}, \ldots, r_k^{\circ}$	initial values of $r_1, \ldots, r_k$
$w_1, \ldots, w_k$	fresh variables

The weakest precondition rule for a function call:

$$\mathrm{WP}(f\ t_1 \ldots t_n, Q) = P_f[\vec{x} \leftarrow \vec{t}] \wedge \forall\, \texttt{result}, w_1, \ldots, w_k.\ Q_f[\vec{x} \leftarrow \vec{t}, \vec{r}^{\circ} \leftarrow !\vec{r}, !\vec{r} \leftarrow \vec{w}] \rightarrow Q[!\vec{r} \leftarrow \vec{w}]$$

Modular proof: when verifying a function call,
we only use the function's contract, not its code.

Examples

```
let max (x y : int) : int
  requires { true }
  ensures  { result >= x /\ result >= y }
  ensures  { result =  x \/ result =  y }
= if x >= y then x else y
```

```
val r : ref int (* declares a global reference *)

let add42 () : int
  requires { true }
  writes   { r }
  ensures  { !r = old !r + 42 /\ result = old !r }
= let v = !r in
  r := v + 42;
  v
```

5. Total correctness: termination

Termination

Goal: prove that the program terminates for every initial state that satisfies the precondition.

It suffices to show that

- every loop makes a finite number of iterations
- recursive function calls cannot go on indefinitely

Solution: prove that every loop iteration and every recursive call decreases a certain value, called variant, with respect to some well-founded order.

For example, for signed integers, a practical well-founded order is

$$i \prec j \quad = \quad i < j \wedge 0 \leq j$$

Loop termination

A new annotation:

$$
\begin{array}{rcl}
e & ::= & \dots \\
 & | & \texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ t \cdot \prec\ \texttt{do}\ e\ \texttt{done}
\end{array}
$$

The corresponding weakest precondition rule:

$$
\begin{array}{l}
\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ s \cdot \prec\ \texttt{do}\ e\ \texttt{done}, Q) = \\
\quad J \wedge \\
\quad \forall w_1, \dots, w_k. \\
\qquad (J \wedge t \rightarrow \mathrm{WP}(e, J \wedge s \prec s[!\vec{r} \leftarrow \vec{w}]))[!\vec{r} \leftarrow \vec{w}] \wedge \\
\qquad (J \wedge \neg t \rightarrow Q)[!\vec{r} \leftarrow \vec{w}]
\end{array}
$$

$r_1, \dots, r_k$ references modified in e

$w_1, \dots, w_k$ fresh variables

Find appropriate variants:

```
let i = ref 0 in
while !i <= 100 do
  variant { ??? }
  i := !i + 1
done
```

```
let sum = ref 1 in
let count = ref 0 in
while !sum <= n do
  invariant { ??? }
  variant   { ??? }
  count := !count + 1;
  sum := !sum + 2 * !count + 1
done
```

A new contract clause:

$$
\begin{array}{l}
\texttt{let rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n)\ :\ \tau \\
\qquad \texttt{requires}\ P_f \\
\qquad \texttt{variant}\ s \cdot \prec \\
\qquad \texttt{writes}\ r_1, \ldots, r_k \\
\qquad \texttt{ensures}\ Q_f \\
\quad = e_f
\end{array}
$$

For each recursive call of f in e:

$$
\begin{array}{rcl}
\mathrm{WP}(f\ t_1 \ldots t_n, Q) & = & P_f[\vec{x} \leftarrow \vec{t}] \wedge s[\vec{x} \leftarrow \vec{t}] \prec s[!\vec{r} \leftarrow \vec{r}^{\circ}] \wedge \\
& & \forall\, \texttt{result}, w_1, \ldots, w_k. \\
& & \quad Q_f[\vec{x} \leftarrow \vec{t}, \vec{r}^{\circ} \leftarrow !\vec{r}, !\vec{r} \leftarrow \vec{w}] \rightarrow Q[!\vec{r} \leftarrow \vec{w}]
\end{array}
$$

Mutual recursion

Mutually recursive functions must have

- their own variant terms
- a common well-founded order

Thus, if f calls $g\ t_1 \ldots t_n$, the variant decrease precondition is

$$s_g[\vec{x}_g \leftarrow \vec{t}\,] \prec s_f[\,!\vec{r}_f \leftarrow \vec{r}_f^{\,\circ}]$$

$\vec{x}_g$	the formal parameters of g
s_f, s_g	the variants of f and g, respectively
$\vec{r}_f$	global references affected by f

6. WHYML types

WhyML supports most of the OCaml types:

- polymorphic types

```
type set 'a
```

- tuples:

```
type poly_pair 'a = ('a, 'a)
```

- records:

```
type complex = { re : real; im : real }
```

- variants (sum types):

```
type list 'a = Cons 'a (list 'a) | Nil
```

To handle algebraic types (records, variants):

- access to record fields:

```
let get_real (c : complex) = c.re
let use_imagination (c : complex) = im c
```

- record updates:

```
let conjugate (c : complex) = { c with im = - c.im }
```

- pattern matching (no `when` clauses):

```
let rec length (l : list 'a) : int variant { l } =
  match l with
  | Cons _ ll -> 1 + length ll
  | Nil -> 0
  end
```

Abstract types must be axiomatized:

```
theory Map
  type map 'a 'b

  function ([])   (a: map 'a 'b) (i: 'a): 'b
  function ([<-]) (a: map 'a 'b) (i: 'a) (v: 'b): map 'a 'b

  axiom Select_eq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 = k2  -> m[k1 <- v][k2] = v

  axiom Select_neq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 <> k2 -> m[k1 <- v][k2] = m[k2]
end
```

Abstract types (cont.)

Abstract types must be axiomatized:

```
theory Set
  type set 'a
  predicate mem 'a (set 'a)

  predicate (==) (s1 s2: set 'a) =
    forall x: 'a. mem x s1 <-> mem x s2
  axiom extensionality:
    forall s1 s2: set 'a. s1 == s2 -> s1 = s2

  predicate subset (s1 s2: set 'a) =
    forall x: 'a. mem x s1 -> mem x s2
  lemma subset_refl: forall s: set 'a. subset s s

  constant empty : set 'a
  axiom empty_def: forall x: 'a. not (mem x empty)
  ...
```

Logical language of WhyML

- the same types are available in the logical language
- `match-with-end`, `if-then-else`, `let-in`
  are accepted both in terms and formulas
- functions et predicates can be defined recursively:

```
predicate mem (x: 'a) (l: list 'a) = match l with
  Cons y r -> x = y \/ mem x r | Nil -> false end
```

  no `variant`s, Why3 requires structural decrease

- inductive predicates (useful for transitive closures):

```
inductive sorted (l: list int) =
  | SortedNil: sorted Nil
  | SortedOne: forall x: int. sorted (Cons x Nil)
  | SortedTwo: forall x y: int, l: list int.
      x <= y -> sorted (Cons y l) ->
        sorted (Cons x (Cons y l))
```

7. Ghost code

Ghost code: example

Compute a Fibonacci number using a recursive function in $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires {                                                     }
  ensures  {
                                                                 }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Ghost code: example

Compute a Fibonacci number using a recursive function in $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) /\
                                              result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Ghost code: example

Instead of an existential we can use a ghost parameter:

```
let rec aux (a b n: int) (ghost k:  int): int
  requires { 0 <= n }
  requires { 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1) (k+1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n 0
```

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

We must be able to eliminate the ghost code
from a program without changing its outcome

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

We must be able to eliminate the ghost code
from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k + 1)$ is ghost, too

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

We must be able to eliminate the ghost code
from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify normal data
  - if r is a normal reference, then $r := \texttt{ghost}\ k$ is forbidden

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

We must be able to eliminate the ghost code
from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k + 1)$ is ghost, too
- ghost code cannot modify normal data
  - if r is a normal reference, then $r := \texttt{ghost}\ k$ is forbidden
- ghost code cannot alter the control flow of normal code
  - if c is ghost, then $\texttt{if}\ c\ \texttt{then}\ \ldots$ and $\texttt{while}\ c\ \texttt{do}\ \ldots\ \texttt{done}$ are ghost

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

We must be able to eliminate the ghost code
from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k + 1)$ is ghost, too
- ghost code cannot modify normal data
  - if r is a normal reference, then `r := ghost k` is forbidden
- ghost code cannot alter the control flow of normal code
  - if c is ghost, then `if c then ...` and `while c do ... done` are ghost
- ghost code cannot diverge
  - we can prove `while true do () done ; assert { false }`

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

Ghost code in WhyML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;   (* get from head *)
                  tail: list 'a;   (* add to tail *)
            ghost elts: list 'a;   (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

Ghost code in WhyML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;   (* get from head *)
                  tail: list 'a;   (* add to tail *)
            ghost elts: list 'a;   (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- local variables and functions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

Ghost code in WhyML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;   (* get from head *)
                  tail: list 'a;   (* add to tail *)
            ghost elts: list 'a;   (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- local variables and functions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

- program expressions

```
let x = ghost qu.elts in ...
```

Lemma-functions

General idea: a function $f\ x$ `requires` P_f `ensures` Q_f that

- returns `unit`
- has no side effects
- terminates

provides a constructive proof of $\forall x : P_f \rightarrow Q_f$

$\Rightarrow$ a pure recursive function simulates a proof by induction

Lemma-functions

General idea: a function $f\ x$ `requires` P_f `ensures` Q_f that

- returns `unit`
- has no side effects
- terminates

provides a constructive proof of $\forall x : P_f \rightarrow Q_f$

$\Rightarrow$ a pure recursive function simulates a proof by induction

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
= match l with Cons a ll -> length_rev_append ll (Cons a r)
             | Nil -> () end
```

Lemma-functions

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
= match l with Cons a ll -> length_rev_append ll (Cons a r)
             | Nil -> () end
```

- by the postcondition of the recursive call:

  ```
  length (rev_append ll (Cons a r)) = length ll + length (Cons a r)
  ```

- by definition of `rev_append`:

  ```
  rev_append (Cons a ll) r = rev_append ll (Cons a r)
  ```

- by definition of `length`:

  ```
  length (Cons a ll) + length r = length ll + length (Cons a r)
  ```

8. Mutable data

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`
- cannot be stored in recursive structures: no `list (ref 'a)`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`
- cannot be stored in recursive structures: no `list (ref 'a)`
- cannot be stored under abstract types: no `set (ref 'a)`

The problem of alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* write/write alias *)
  assert { !s = 1 /\ !s = 2 }   (* in fact, !s = 3 *)
```

The problem of alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;    (* write/write alias *)
  assert { !s = 1 /\ !s = 2 }    (* in fact, !s = 3 *)
```

```
val g : ref int

let set_from_g (r: ref int): unit writes {r}
  ensures { !r = !g + 1 }
= r := !g + 1

let wrong () =
  set_from_g g;    (* read/write alias *)
  assert { !g = !g + 1 }    (* contradiction *)
```

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- Why3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- WHY3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The user must indicate the external dependencies of abstract functions:

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- otherwise the static control of aliases does not have enough information

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- WHY3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The user must indicate the external dependencies of abstract functions:

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- otherwise the static control of aliases does not have enough information

For programs with arbitrary pointers we need more sophisticated tools

- memory models (for example, “address-to-value” arrays)
- handle aliases in the VC: separation logic, dynamic frames, etc.

Aliasing restrictions in WHYML

⇒ certain structures cannot be implemented

Still, we can specify them and verify the client code

```
type array 'a model { mutable elts: map int 'a;
                              length: int }
invariant { 0 <= self.length }
```

- fields `length` et `elts` can only be used in annotations (model type)
- all access is done via abstract functions
- the type invariant is verified at the boundaries of function calls
  - WHY3 implicitly adds the necessary pre- et postconditions

Abstract specification

```
type array 'a model { mutable elts: map int 'a;
                                length: int }
invariant { 0 <= self.length }

val ([]) (a: array 'a) (i: int): 'a
  requires { 0 <= i < a.length }
  ensures  { result = a.elts[i] }

val ([]<-) (a: array 'a) (i: int) (v: 'a): unit writes {a}
  requires { 0 <= i < a.length }
  ensures  { a.elts = (old a.elts)[i <- v] }

val length (a: array 'a): int ensures { result = a.length }

function get (a: array 'a) (i: int): 'a = a.elts[i]
```

- the immutable fields are preserved — implicit postcondition
- the logical function `get` has no precondition
    - its result outside of the array bounds is undefined

9. Modular programming considered useful

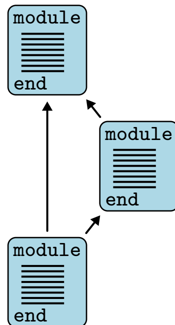
Declarations

- types
  - abstract: `type t`
  - synonym: `type t = list int`
  - variant: `type list 'a = Nil | Cons 'a (list 'a)`
- functions / predicates
  - uninterpreted: `function f int: int`
  - defined: `predicate non_empty (l: list 'a) = l <> Nil`
  - inductive: `inductive path t (list t) t = ...`
- axioms / lemmas / goals
  - `goal G: forall x: int, x >= 0 -> x*x >= 0`
- program functions (*routines*)
  - abstract: `val ([]) (a: array 'a) (i: int): 'a`
  - defined: `let mergesort (a: array elt): unit = ...`
- exceptions
  - `exception Found int`

Modules

Declarations are organized in modules

- purely logical modules are called theories

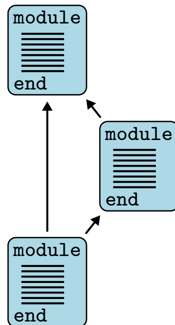


Declarations are organized in modules

- purely logical modules are called theories

A module M_1 can be

- used (`use`) in a module M_2
  - symbols of M_1 are shared
  - axioms of M_1 remain axioms
  - lemmas of M_1 become axioms
  - goals of M_1 are ignored



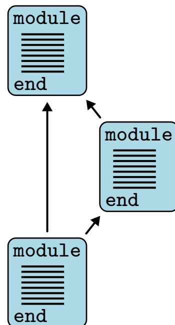
Modules

Declarations are organized in modules

- purely logical modules are called theories

A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2
    - declarations of M_1 are copied or instantiated
    - axioms of M_1 remain axioms or become lemmas
    - lemmas of M_1 become axioms
    - goals of M_1 are ignored



Modules

Declarations are organized in modules

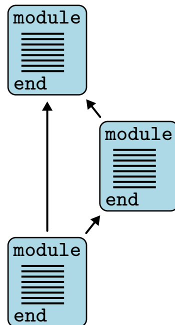
- purely logical modules are called theories

A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2

Cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function
- (*not yet, but easy to do*) a `val` with a `let`



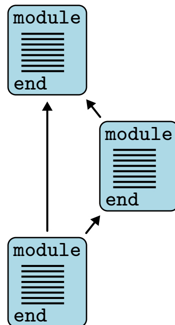
Modules

Declarations are organized in modules

- purely logical modules are called theories

A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2



Cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function
- (*not yet, but easy to do*) a `val` with a `let`

One missing piece coming soon:

- instantiate a used module with another module